Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Bac

Pretvorba PL/SQL izvorne kode v graf prehodov aplikacije

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: doc. dr. Boštjan Slivnik

Ljubljana 2015

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Opis:

Poiščite in realizirajte način pretvorbe izvorne kode spletne aplikacije, ki je napisana v programskem jeziku PL-SQL, v graf prehodov aplikacije. Svojo rešitev preizkusite na primeru izvorne kode spletne aplikacije eŠtudent.

Opis (angleški):

Design and implement a transformation of a web application source code written in PL-SQL programming language into the application transition graph. Test and evaluate your solution by applying it to eŠtudent source code.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Luka Bac, z vpisno številko **63090412**, sem avtor diplomskega dela z naslovom:

Pretvorba PL/SQL izvorne kode v graf prehodov aplikacije

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Boštjana Slivnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 11. marca 2015 Podpis avtorja:

Zahvalil bi se rad mentorju Doc. Dr. Boštjanu Slivniku, ki se je tekom izdelave naloge zelo zavzel za korektnost, strokovnost in natančnost. Prav tako gre zahvala tudi vsem, ki so me v času mojega študija podpirali - predvsem staršem, bratu in prijateljem. Njihova moralna podpora, prijaznost in potrpežljivost so mi bila vseskozi v oporo.

Zorz.

V spomin čudovitemu očetu, ki nas je vse navdal z globokim vedoželjem.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SQL	Structured Query Language	strukturirani povpraševalni jezik za delo s podatkovnimi bazami
PL/SQL	Procedural Language / SQL	proceduralna razširitev za SQL
HTML	HyperText Markup Language	jezik za označevanje nadbesedila
JS	Javascript	programski jezik Javascript
ASM	Atomic Section Model	Atomični Model Odseka
ATG	Application Transition Graph	Graf Prehodov Aplikacije
CIM	Component Interaction Model	Model Interakcij Komponent
Flex	Fast lexical analyzer	Hitri leksikalni analizator

Povzetek

Pri obratnem inženiringu aplikacij je analitiku v pomoč pregledni graf funkcijskih klicev aplikacije. Prikazali smo delovanje razvite aplikacije, ki avtomatsko generira tak graf. Za njegovo predstavitev smo uporabili atomični model odseka (ASM) s približkom grafa prehodov aplikacije (ATG). Aplikacija je bila razvita na primeru informacijskega sistema e-Študent. Sistem temelji na podatkovni bazi proizvajalca Oracle, na voljo nam je bila izvozna datoteka baze, iz katere je bilo treba pridobiti izvorne datoteke s funkcijami, dinamičnimi strukturami in procedurami v jeziku PL/SQL.

Najprej je bilo treba bazo restavrirati, nato pa iz nje izvleči izvorno kodo. Za to kodo smo nato z odprtokodnim orodjem *flex* definirali potrebno leksikalno zgradbo jezika PL/SQL - v tem primeru predvsem opis funkcijskih klicev. Nato smo z uporabo dobljenega leksikalnega analizatorja generirali s programskim jezikom *Python* tekstovno datoteko v formatu za opis grafov DOT. To datoteko smo z orodji iz paketa *Graphviz* pretvorili v grafe, ki nakazujejo, katere datoteke kličejo katere funkcije, kar je v pomoč pri razumevanju strukture sistema. Te grafe smo poskusili narediti bolj berljive z dodatkom funkcij Javascript in stilov CSS za obarvanje povezav ter z izračunom vhodne in izhodne stopnje vozlišč, s pomočjo katere lahko izločimo posamezna vozlišča.

Ključne besede: PL/SQL, Oracle SQL, atomični model odseka, graf prehodov aplikacije, model interakcij komponent, Flex, Graphviz, Python.

Abstract

A graph providing an overview of an application would serve the reverse engineering effort well. We have walked through the workings of an application that was developed to automatically generate one such graph, which was represented by an Atomic Section Model with an approximation of its Application Targeted Graph. The case-study for this application was the student information system e-Študent. This system uses an Oracle Database to host the data, while our effort only had access to a database dump from which to retrieve the PL/SQL source code in the form of functions, dynamic structures and procedures.

The first step was to restore the database itself, followed by scraping the source code into text files which were then analyzed using a lexical analyzer developed by the open source tool *flex*, in which we defined the required PL/SQL lexical structure, focused mainly on recognizing function calls. This output was then fed through a Python script that generated a file in the DOT format, describing the function calls of the individual input files. This was presented by rendering it into a graph with the open source graph visualization software *Graphviz*. These graphs were then made easier to read by injecting Javascript functions and CSS styles to enable highlighting of edges and vertices. In addition, the application calculates the in and out degree of each vertex, which can help the analyst in deciding to exclude certain vertices.

Keywords: PL/SQL, Oracle SQL, atomic section model, application transition graph, component interaction model, Flex, Graphviz, Python.

Poglavje 1

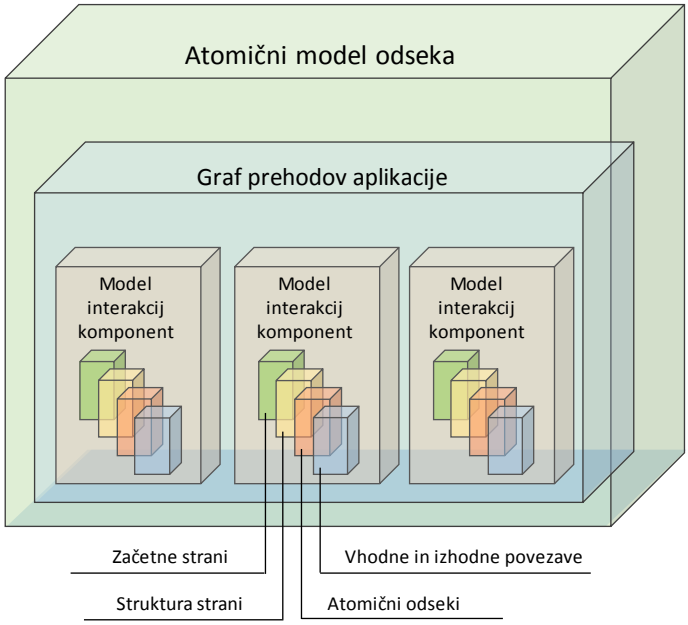
Uvod

Cilj obratnega inženirstva je priti do globljega poznavanja subjekta, o katerem ni na voljo natančne specifikacije. To nato omogoči replikacijo subjekta, njegovo nadgradnjo, popravo ali integracijo z obstoječimi sistemi ter formulacijo manjkajoče specifikacije.

Spletne aplikacije, kot aplikacije kateregakoli drugega tipa, lahko zastarijo, prvotni razvijalci aplikacije odidejo, znanje se tako ali drugače porazgubi. Za vpogled v delovanje aplikacij in pridobivanje tega izgubljenega znanja lahko uporabimo obratno inženirstvo. Pri tem bi bil v veliko pomoč pogled na aplikacijo od zgoraj navzdol, kar lahko dosežemo z gradnjo enega splošnega ali več specifičnih grafov iz izvorne kode same.

Eden od načinov predstavitve takih grafov je z atomičnim modelom odseka (angl. Atomic Section Model - ASM)[4], ki je predstavljen z grafom prehodov aplikacije (angl. Application Transition Graph - ATG). ATG je usmerjen graf, ki je posledično sestavljen iz modelov interakcij komponent (angl. Component Interaction Model - CIM), njihovih povezav, spremenljivk, ki hranijo vsa možna stanja predstavitvene plasti, in množice vseh začetnih strani. Posamezen element CIM pa vsebuje množico začetnih strani, ki ga naslavljajo, množico atomičnih odsekov, ki ga naslavljajo, izraz, ki opisuje strukturo strani, ter množico vhodnih in izhodnih povezav [slika 1.1]. V tej diplomski nalogi se bomo predvsem osredotočili na avtomatsko generiranje približka ATG, ki bo vseboval povezave med posameznimi vozlišči.

Tovrstne spletne aplikacije so lahko napisane z uporabo različnih tehnologij,



Slika 1.1: Prikaz elementov atomičnega modela odseka.

od programskega jezika PHP in njegovih ogrodij kot sta Zend in Symfony, do programskega jezika C# in ogrodja ASP.NET ter programskega jezika Python in ogrodji kot sta Zope in Django, in podobno. Ta diplomska naloga se bo osredotočila na obratno inženirstvo spletnih aplikacij, ustvarjenih z Oracle jezikom PL/SQL, ki s klici procedur in funkcij dinamično ustvari stran v jeziku HTML. Primer take procedure PL/SQL, ki uporablja klic funkcije htp.p(rint) za izpis kode HTML je naslednji:

```
1  PROCEDURE         "PROC_EXCEPTION_IZPIS"
2  (
3      v_imeProcedure IN VARCHAR2,
4      v_oraError IN VARCHAR2
5  )
6  as
7    v_adminMail VARCHAR2(100);
8  begin
9
10   v_adminMail := FUN_PAR('ADMIN_MAIL');
11
12   htp.p('<CENTER>');
```

```
13    htp.p('<table border="0" cellpadding="0" cellspacing
         ="0" width="350"><tr><td><img border="0" src="/
         images/e-student/pika.gif" width="1" height="1"
         vspace="5"></td></tr></table>
14    <table border="0" cellpadding="0" cellspacing="0"
         width="350">
15    <tr>
16      <td bgcolor="#800000" width="1"><img border="0" src
           ="/images/e-student/pika.gif" width="1" height
           ="1"></td>
17      <td>
18        <table border="0" cellpadding="0" cellspacing="0"
             width="100%">
19          <tr>
20            <td width="100%" bgcolor="#800000"> <font
                 class="oknotext">Napaka</font></td>
21          </tr>
22          <tr>
23            <td width="100%" bgcolor="#800000"><img border
                 ="0" src="/images/e-student/pika.gif" width
                 ="1" height="1"></td>
24          </tr>
25          <tr>
26            <td width="100%">
27              <table border="0" cellpadding="3"
                   cellspacing="0" width="100%">
28                <tr>
29                  <td width="100%"><font class="
                       izbrano_text">V sklopu '||
                       v_imeProcedure||' je pri<9a>lo do
                       napake.<br><font color="#ff0000">'||
                       v_oraError||'</font><br>Posvetujte se
                        s <a href="mailto:'|| v_adminMail
                       ||'">skrbnikom</a> sistema.</font></
                       td>
30                </tr>
31              </table>
32            </td>
33          </tr>
34          <tr>
35            <td width="100%" bgcolor="#800000"><img border
                 ="0" src="/images/e-student/pika.gif" width
                 ="1" height="1"></td>
36          </tr>
37        </table>
```

```
38      </td>
39      <td bgcolor="#800000" width="1"><img border="0" src
           ="/images/e-student/pika.gif" width="1" height
           ="1"></td>
40    </tr>
41    </table>');
42
43    htp.p('</CENTER>');
44
45  exception
46  when others then
47      htp.nl;
48      htp.p('<font class="errortext">Napaka v sklopu
           PROC_exception_izpis.</font>');
49  end;
```

Najprej je takšno kodo treba dobiti, če je še nimamo, zato bo prvo poglavje namenjeno pridobivanju izvorne kode iz izvozne datoteke in restavriranju podatkovne baze. V naslednjem poglavju bo opisana uporaba odprto kodnega orodja *flex*, s katerim se definira potrebna leksikalna zgradba PL/SQL - v tem primeru predvsem opis funkcijskih klicev. Nato bo opisano, kako iz izhoda dobljenega leksikalnega analizatorja s programskim jezikom *Python* generiramo tekstovno datoteko v formatu za opis grafov DOT. Ta datoteka DOT se z orodji iz paketa *Graphviz* pretvori v grafe, ki so približki ATG in kjer bo jedro posameznega elementa CIM vsebovalo le povezave elementov, ki se sklicujejo nanj, ter povezave na elemente, ki jih ta element sam kliče.

Poglavje 2

Pridobivanje izvorne kode

Najprej je treba pridobiti izvorno kodo PL/SQL v tekstovnih datotekah. Če imamo srečo, da je originalna baza še postavljena in deluje, lahko iz nje izvlečemo izvorno kodo in jo zapišemo v tekstovne datoteke, te pa naknadno obdelamo. V nasprotnem primeru, če imamo na voljo le izvozno datoteko (angl. database dump), je treba bazo najprej restavrirati. V našem testnem primeru baze e-Študent smo dobili le datoteko vz64.dmp, ki je bila narejena z različico *Oracle Database 9i*.

2.1 Restavriranje baze iz izvozne datoteke

Preden lahko restavriramo stanje iz izvozne datoteke, je seveda treba naložiti podatkovno bazo. Tako smo naložili 64-bitno različico podatkovne baze *Oracle Database Express Edition 11g Release 2*. Ko je baza naložena, zaženemo preko ukazne vrstice SQL*Plus in se povežemo na bazo kot uporabnik SYSTEM, ki je bil ustvarjen ob naložitvi podatkovne baze.

```
1 connect SYSTEM as SYSDBA
```

Z imperativom *as SYSDBA* povemo, da se želimo povezati z administratorskimi pravicami, ki jih bomo potrebovali za nekatere nadaljnje ukaze. Nato ustvarimo shemo, v katero bomo restavrirali bazo. Pri bazi Oracle se z ustvaritvijo uporabnika ustvari tudi istoimenska shema. Temu uporabniku seveda dodelimo geslo (*password*), nato mu dodamo še pravice, ki jih bo potreboval za dostop do baze (*CONNECT*), za ustvarjanje tabel in drugih struktur (*RESOURCE*), za

uvoz baze (*IMP_FULL_DATABASE*) in za uporabo sistemskih knjižnic (*EXECUTE_CATALOG_ROLE*, *SELECT_CATALOG_ROLE*).

```
1 CREATE USER VZ64 IDENTIFIED BY password;
2 GRANT CONNECT TO VZ64;
3 GRANT RESOURCE TO VZ64;
4 GRANT IMP_FULL_DATABASE TO VZ64;
5 GRANT EXECUTE_CATALOG_ROLE TO VZ64;
6 GRANT SELECT_CATALOG_ROLE TO VZ64;
```

S tem sta uporabnik in shema pripravljena za uvoz podatkov. Na tem mestu ne preostane nič drugega, kot da poskusimo z uvozom. Različice od *Oracle 10g* naprej omogočajo in priporočajo uporabo orodij *expdp* in *impdp* za izvoz in uvoz baze namesto starih orodij *exp* in *imp*, ki sta bili na voljo prej. Ta nabor novih orodij je hitrejši, bolj zmogljiv in bolj fleksibilen; uvozni del bi sam na primer poskrbel za to, da bi ustvaril potrebne uporabnike in sheme, kar smo v prejšnjem koraku storili sami. Ker je bil v našem testnem primeru izvoz baze narejen z različico *Oracle 9i* in tako s starim orodjem *exp*, moramo tudi mi narediti uvoz s starim orodjem *imp*, saj sta formata nezdružljiva.

Pri nas se je hitro izkazalo, da je ob poskusu uvoza prišlo do napak - izvoz je bil namreč narejen v znakovnem naboru *EE8MSWIN1250*. *Oracle 9i* je bil na voljo v dveh različicah, univerzalni in zahodnoevropski, slednja se je avtomatsko naložila z znakovnim naborom *EE8MSWIN1250*. *Oracle 11g* pa je na voljo le v univerzalni različici, ki ima privzet znakovni nabor *AL32UTF8*. Na neskladje znakovnih naborov nas program za uvažanje opozori. Če poskusimo uvoziti s temi nastavitvami, naletimo na napake naslednjega tipa:

```
ORA-12899: value too large for column "VZ64"."APP"."
   ZG_MEJA" (actual: 31, maximum: 30)
```

Najprej smo poskusili napako zaobiti z nastavitvijo NLS_LENGTH_SEMANTICS na CHAR, kar bi pomenilo, da bi baza računala dolžino niza glede na število znakov, in ne glede na število bajtov, vendar to ni obrodilo sadov. Za pravilno rešitev se je izkazala zamenjava znakovnega nabora podatkovne baze. Naslednje ukaze seveda zopet izvajamo z uporabnikom z administratorskimi pravicami, hkrati pa uporabimo omejeno sejo, da se prepričamo, da ni povezanih drugih uporabnikov.

```
1 SHUTDOWN IMMEDIATE;
2 STARTUP MOUNT;
```

```
3 ALTER SYSTEM ENABLE RESTRICTED SESSION;
4 ALTER DATABASE OPEN;
5 ALTER DATABASE CHARACTER SET INTERNAL_USE EE8MSWIN1250;
6 SHUTDOWN;
7 STARTUP RESTRICT;
8 SHUTDOWN;
9 STARTUP;
```

Hkrati se moramo prepričati, da ima shema dodeljenega dovolj prostora za uvoz podatkov, sicer lahko pride do naslednjih napak:

```
ORA-01653: unable to extend table SYS.SOURCE$ by 128 in
   tablespace SYSTEM
```

V našem primeru smo najprej poskušali shemi vklopiti funkcijo *autoextend*, ki bi načeloma morala sama poskrbeti za to, da ima na voljo dovolj prostora, vendar je to delovalo šele, ko smo ročno povečali prostor. Najprej pogledamo, kje so datoteke tabelnih prostorov (angl. tablespace), nato jih povečamo in priredimo uporabniku, če mu še niso bile prirejene, z naslednjimi ukazi:

```
1 SELECT tablespace_name, file_name FROM dba_data_files;
2 ALTER DATABASE DATAFILE 'C:\oraclexe\app\oracle\oradata\
   XE\USERS.DBF' RESIZE 500M;
3 ALTER USER VZ64 DEFAULT TABLESPACE users;
```

Po teh spremembah uvoz za naše potrebe deluje. Kot rečeno uporabimo orodje *imp*, ter mu z argumenti povemo, katero datoteko želimo uvoziti in v katero shemo jo želimo uvoziti ter da želimo opis postopka in morebitne napake shraniti v tekstovno datoteko.

```
1 imp -file="vz64.dmp" -fromuser=VZ64 -touser=VZ64 -log=
   import.log
```

Zdaj so v podatkovni bazi vsi podatki, funkcije, procedure in paketi, ki so bili definirani v izvozni datoteki. Nekateri prožilci (angl. trigger), omejitve (angl. constraint), drugi tabelni prostori in uporabniki se sicer še vedno niso prenesli in tudi povezave na druge podatkovne baze seveda ne delujejo. Vendar nas ta del restavriranja baze v našem primeru ne zanima, saj imamo v bazi vse podatke, ki jih potrebujemo, in je ta del restavriranja baze zunaj obsega tega diplomskega dela.

2.2 Izpis izvorne kode iz baze

Ni dovolj, da so podatki le v bazi - nekako jih je treba še pridobiti iz nje. Izvorno kodo posamezne funkcije in procedure lahko dobimo v tabeli *dba_source* z ukazom:

```
1 SELECT text FROM dba_source WHERE name='
    PROC_GENERIRAJ_ROK_VSI' AND TYPE in ('FUNCTION', '
    PROCEDURE') ORDER BY line;
```

Na podoben način lahko z uporabo funkcije *DBMS_METADATA.GET_DDL* dobimo izvorno kodo paketov in dinamičnih struktur, izbiramo pa iz posebne navidezne tabele *DUAL*, ker sintaksa Oracle SQL zahteva, da vedno izbiramo iz neke tabele:

```
1 SELECT DBMS_METADATA.GET_DDL('PACKAGE_BODY', '
    DYN_ABOUT_BOX') FROM dual;
```

Naše delo bi zaradi preglednosti olajšalo to, da bi bila vsaka funkcija, procedura in dinamična struktura v svoji tekstovni datoteki. Zato smo napisali skripto, ki združuje več funkcij in nam omogoča izpis izvorne kode vseh funkcij, procedur in paketov, katerih lastnik je izbrani uporabnik.

Za izpis izvorne kode funkcij in procedur z ukazom *SPOOL* smo napisali funkcijo v datoteko scrapeEntry.sql:

```
1 SPOOL &program_dir\output\&1..txt
2 SELECT text FROM dba_source WHERE name='&1' AND TYPE in
    ('FUNCTION', 'PROCEDURE') ORDER BY line;
3 SPOOL OFF;
```

Za izpis izvorne kode paketov pa funkcijo v datoteki scrapePackage.sql:

```
1 SPOOL &program_dir\output\&1..txt
2 SELECT DBMS_METADATA.GET_DDL('PACKAGE_BODY', '&1') FROM
    dual;
3 SPOOL OFF;
```

Nato smo napisali krovno funkcijo allFiles.sql, ki kliče ti dve funkciji:

```
1 DEFINE program_dir = C:\Users\Luka\Desktop
2 SET VERIFY OFF
3
4 PROMPT ======== creating subscript ========
5
6 -- turn off SQL feedback
7 SET FEEDBACK OFF
8 SET PAGESIZE 0      -- disable header/feedback/bla
```

```
 9 SET LINESIZE 32767  -- set linelength to max
10 SET LONG 999999999  -- Sets maximum width (in bytes) for
       displaying CLOB, LONG, NCLOB and XMLType values; and
       for copying LONG values
11 SET TRIMSPOOL ON    -- trim trailing spaces
12
13 SPOOL &program_dir\subscript.sql    -- redirect all
      output to file
14
15 SELECT DISTINCT '@&program_dir\scrapeEntry', object_name
16 FROM dba_objects
17 WHERE object_type IN ('FUNCTION', 'PROCEDURE')
18         AND OWNER='&1';
19
20 SELECT DISTINCT '@&program_dir\scrapePackage',
      object_name
21 FROM dba_objects
22 WHERE object_type IN ('PACKAGE')
23     AND OWNER='&1';
24
25 SPOOL OFF;
26
27 @&program_dir\subscript.sql
28
29 SET FEEDBACK 6
30 SET PAGESIZE 66
```

V tej funkciji smo najprej s spremenljivko *program_dir* definirali mapo, v kateri so datoteke funkcij SQL, ki jih bomo klicali, vključno z allFiles.sql. Na to spremenljivko se nato sklicujeta tudi funkciji scrapeEntry.sql in scrapePackage.sql. Ker ukaz *SPOOL* v bistvu preusmeri tekst rezultata naših stavkov SQL v datoteko, je treba ta tekst najprej pravilno oblikovati, da bomo res dobili le izvorno kodo brez povratne informacije ukazov. Zato izklopimo vse tovrstne nastavitve, nastavimo dolžino vrstice in LONG tipov na največje možne ter vklopimo nastavitev za odstranitev odvečnih presledkov. Ko smo to storili, zapišemo dejanske ukaze za posamezno funkcijo, proceduro in paket v vmesno datoteko *subscript.sql*, ki je dejansko le nabor klicev funkcij scrapeEntry ali scrapePackage za posamezen element:

```
@C:\Users\Luka\Desktop\scrapeEntry
   PROC_GENERIRAJ_ROK_VSI
```

Na koncu ta funkcija izvede to vmesno datoteko.

S tako pripravljenimi skriptami nam preostane le, da v SQL*Plus kličemo krovno funkcijo allFiles.sql z argumentom uporabnika, ki je lastnik funkcij, procedur in paketov, ki jih želimo izvleči iz baze.

```
@C:\Users\Luka\allFiles VZ64
```

Tako se v mapi *output* ustvarijo tekstovne datoteke z izvorno kodo.

Poglavje 3

Format izhodnih podatkov

3.1 Leksikalna zgradba jezika PL/SQL

Izvorne datoteke vsebujejo kodo PL/SQL. PL/SQL (angl. Procedural Language/SQL) je razširitev jezika SQL, ki omogoča uporabo konstruktov postopkovnih jezikov, kot so spremenljivke, pogojni stavki in zanke v obliki funkcij, procedur in dinamičnih struktur. Ker je treba kodo PL/SQL razčleniti in priti do funkcijskih klicev, je seveda najprej treba poznati leksikalno zgradbo jezika - katere leksikalne enote ima in kako so znakovno predstavljene. Tabeli 3.1 in 3.2 prikazujeta posamezne simbole in njihov pomen v jeziku. Poleg teh simbolov so, kot v klasičnem jeziku SQL, uporabljeni tudi različni literali za tekst, števila, datume, časovne intervale; PL/SQL pa ponuja še razširitev za logične vrednosti (**TRUE, FALSE, NULL**).

3.2 Flex

Flex[1] (angl. fast lexical analyzer) je odprtokodno orodje, ki iz podane gramatike generira leksikalni analizator, ki za posamezno pravilo izvrši uporabniško kodo. Tak leksikalni analizator, generiran z verzijo flex 2.5.35, smo uporabili za izluščenje funkcijskih klicev iz izvorne kode, vendar je bilo najprej seveda treba prej opisano leksikalno zgradbo pretvoriti v sintakso, ki jo bo flex razumel, da bo lahko ustvaril naš analizator.

Simbol	Pomen
+	operator seštevanja
:=	operator prirejanja
=>	operator asociacije
%	atribut
’	začetek/konec znakovnega niza
.	komponente
||	operator konkatenacije
/	operator deljenja
**	operator potenciranja
(	začetek izraza ali seznama
)	konec izraza ali seznama
:	indikator spremenljivke gostitelja
,	ločilo med elementi
<<	začetek oznake
>>	konec oznake
/*	začetek več-vrstičnega komentarja
*/	konec več-vrstičnega komentarja

Tabela 3.1: Prvi del simbolov jezika PL/SQL in njihovi pomeni

Simbol	Pomen
*	operator množenja
”	začetek/konec identifikatorja med narekovaji
..	operator razpona
=	relacijski operator enakosti
<>	relacijski operator neenakosti
!=	relacijski operator neenakosti
~=	relacijski operator neenakosti
^=	relacijski operator neenakosti
<	relacijski operator manjše
>	relacijski operator večje
<=	relacijski operator manjše ali enako
>=	relacijski operator večje ali enako
@	operator oddaljenega dostopa
- -	enovrstični komentar
;	stavčni terminator
-	operator odštevanja ali negacije

Tabela 3.2: Drugi del simbolov jezika PL/SQL in njihovi pomeni

Datoteka Flex je tako razdeljena na tri dele, od katerih je vsak ločen z znakoma %%; ti deli so definicije, pravila in uporabniška koda. Obvezen del so načeloma le pravila, ostala dela pa sta neobvezna. Mi smo uporabili vse tri. Tako se v prvem delu, definicijah, na začetku poda izbor opcij za flex sam, sklice na knjižnice ter definicijo začetnih pogojev, ki bodo uporabljeni pri pravilih. Nato smo vse zgoraj navedene leksikalne elemente opisali z regularnimi izrazi, s čimer smo dosegli, da lahko pri pravilih operiramo s temi leksikalnimi enotami in se nam ni več treba ukvarjati z njihovo dejansko implementacijo. Naš opis jezika PL/SQL za potrebe orodja flex je naslednji:

```
1  %option noyywrap
2
3  %top{
4      #include <strings.h>
5  }
6
7  %s START
8  %s BLOCK
9  %x COMMENT
10
11 PLUS                    \+
12 ASSIGNMENT              :=
13 ASSOCIATION             =>
14 ATTRIBUTE               %
15 STRINGDELIM             '
16 COMPONENT               \.
17 CONCATENATION           \|\|
18 DIVISION                \/
19 EXPONENTIATION          \*\*
20 EXPRESSIONLEFT          \(
21 EXPRESSIONRIGHT         \)
22 HOSTVAR                 :
23 SEPARATOR               ,
24 LABELLEFT               <<
25 LABELRIGHT              >>
26 COMMENTLEFT             \/\*
27 COMMENTRIGHT            \*\/
28 MULTIPLICATION          \*
29 QUOTEDIDENTDELIM        "
30 RANGE                   \.\.
31 EQUALS                  =
32 NOTEQUALS               (<>)|(!=)|(~=)|(\^=)
33 LESSTHAN                <
```

```
34 GREATERTHAN              >
35 LEQ                      <=
36 GEQ                      >=
37 REMOTEACCESS             @
38 SINGLECOMMENT            --
39 TERMINATOR               ;
40 MINUS                    -
41
42 WHITESPACE               [ \n\r\t\f]*
43 IDENTIFIER               [A-Za-z][A-Za-z0-9$#_]*
44 INTEGER                  [+-]?[0-9]+
45 NUMBER                   [+-]?(([0-9]+(\.[0-9]*)?)
      |(\.[0-9]+))([eE]?[+-]?[0-9]+)?(?:[fFdD]?)
46 DATE                     DATE\s
      +'[0-9]{4}-[0-9]{2}-[0-9]{2}'
47 TIMESTAMP                TIMESTAMP\s
      +'[0-9]{4}-[0-9]{2}-[0-9]{2}\s+
      [0-9]{2}:[0-9]{2}:[0-9]{2}\.[0-9]{2}'
48 BOOLEAN                  (?i:true|false|null)
```

Temu sledijo dejanska pravila, katero akcijo naj leksikalni analizator proži, ko naleti na določen, prej definiran, vzorec. Tu so uporabljeni tudi začetni pogoji - predvsem za preklop v stanje, v katerem zavržemo vse, kar je vsebina komentarjev. V tem delu je opisano tudi, kaj bo leksikalni analizator naredil, ko bo prepoznal funkcijski klic. V našem primeru ga le izpiše, saj ga bomo v naslednjih korakih uporabili. Naša pravila so naslednja:

```
49 %%
50 {COMMENTLEFT}         BEGIN(COMMENT);
51 {SINGLECOMMENT}.*
52 {IDENTIFIER}{WHITESPACE}{EXPRESSIONLEFT}.*{
      EXPRESSIONRIGHT}{WHITESPACE}{TERMINATOR} printf("%s\n
      ", yytext);
53 <COMMENT>{COMMENTRIGHT}     BEGIN(INITIAL);
54 <COMMENT>.|\n
55 .|\n
```

Na koncu je še del, ki vsebuje uporabniško kodo. Tu predvsem malo izboljšamo uporabniško izkušnjo in omogočimo ali uporabo programa brez argumentov, kjer bere iz standardnega vhoda, ali pa, da vhod predstavlja datoteka. Naš del z uporabniško kodo je naslednji:

```
56 %%
57 int main(int argc, char* argv[]) {
```

```
58      if (argc == 2) {
59          yyin = fopen(argv[1], "r");
60          yylex();
61          fclose(yyin);
62      } else if (argc == 1) {
63          yylex();
64      } else {
65          printf("This program can be run without an
                argument for standard input, or a filename as
                 an argument.\n");
66          return -1;
67      }
68  }
```

3.3 Graphviz

Graphviz[3] je odprtokodni paket orodij, ki iz datoteke formata DOT generira graf na različne načine. Uporabili smo različico Graphviz 2.38. Najprej definiramo, da gre za usmerjen graf, in mu določimo različne opcije, nato (glede na to ali želimo graf z gručenjem ali ne) poimenujemo podgrafe in v njih povežemo posamezna vozlišča z operatorjem ->. Ko je gručenje vklopljeno, moramo zato, da imamo lahko več vozlišč z enako oznako, tem vozliščem določiti še, kateri gruči pripadajo, ter jim popraviti oznako na funkcijski klic sam. Vozlišče, ki je nevidno in se imenuje enako kot gruča, je interno in dejansko predstavlja gručo samo. Izsek datoteke DOT, ki jo generira naš program, ko je gručenje izklopljeno, je naslednji:

```
1  strict digraph {
2      graph [compound=true];
3      "proc_analiza_polaganja_prof" -> "proc_obroba"
4      "proc_analiza_polaganja_prof" -> "proc_obroba_end"
5      "fun_prijavi_na_kolokvij" -> "fun_parn"
6  }
```

Izsek z gručenjem pa je tak:

```
1  strict digraph {
2      graph [compound=true];
3      subgraph "clusterPROC_VSTAVI_SKLEP" {
4          label = "PROC_VSTAVI_SKLEP"
5          "proc_vstavi_sklep" [label="" style=invis]
6          "proc_vstavi_sklep|proc_izpis_statusa" ->
               "clusterPROC_IZPIS_STATUSA"
7          "proc_vstavi_sklep|proc_izpis_statusa"
               [label="proc_izpis_statusa"]
8      }
9      subgraph "clusterFUN_DATUM_ODDAJE_TEME" {
10         label = "FUN_DATUM_ODDAJE_TEME"
11         "fun_datum_oddaje_teme" [label="" style=invis]
12         "fun_datum_oddaje_teme|proc_izbira_datuma" ->
               "clusterPROC_IZBIRA_DATUMA"
13         "fun_datum_oddaje_teme|proc_izbira_datuma"
               [label="proc_izbira_datuma"]
14     }
15     [...]
16 }
```

Poglavje 4

Združevanje v celoto

Zdaj, ko imamo generiran leksikalni analizator in definirano sintakso datoteke DOT, lahko podatke s katerimkoli programskim jezikom enostavno obdelamo in združimo. Lahko bi si izbrali recimo lupino okolja Linux, vendar smo se v našem primeru odločili za Python[2].

```
1  import os
2  import sys
3  import subprocess
4  import getopt
5
6  verbose = False
7  cluster = False
8  recursive = False
9  pretty = False
10 srcdir = os.getcwd()
11 output = "graph"
12 ext = ".txt"
13 splitter = "|"
14
15 try:
16     opts, args = getopt.getopt(sys.argv[1:], "hcvrps:e:o
          :t:", ["help", "cluster", "verbose", "recursive",
           "pretty", "source=", "extension=", "output=", "
          splitter="])
17 except getopt.GetoptError:
18     print "Valid options: -h --help -c --cluster -v --
          verbose -r --recursive -p --pretty -s --source -e
           --extension -o --output, -t --splitter"
19     sys.exit(1)
```

```
20
21 for opt, arg in opts:
22     if opt in ("-h", "--help"):
23         print "Valid options: -h --help -c --cluster -v
               --verbose -r --recursive -p --pretty -s --
               source -e --extension -o --output"
24         sys.exit(0)
25     elif opt in ("-c", "--cluster"):
26         cluster = True
27     elif opt in ("-v", "--verbose"):
28         verbose = True
29     elif opt in ("-r", "--recursive"):
30         recursive = True
31     elif opt in ("-p", "--pretty"):
32         pretty = True
33     elif opt in ("-s", "--source"):
34         if os.path.isdir(arg):
35             srcdir = arg
36         else:
37             print "Specified source directory does not
                   exit or is not a directory"
38             exit(1)
39     elif opt in ("-e", "--extension"):
40         ext = "." + arg
41     elif opt in ("-o", "--output"):
42         output = arg
43     elif opt in ("-t", "-splitter"):
44         splitter = arg
```

V prvem delu poskrbimo za uvoz knjižnic, nastavimo parametre na privzete vrednosti ter preberemo kateri argumenti so bili podani programu. Prepoznani argumenti so naslednji: -h ali –help za izpis pomoči uporabniku; -c ali –cluster, kjer posamezen element ni več funkcijski klic, temveč skupina funkcijskih klicev, ki jih kliče element; -v ali –verbose za podroben izpis delovanja programa; -r ali –recursive da program upošteva tudi funkcijske klice znotraj funkcijskih klicev; -p ali –pretty da program na koncu dobljeno datoteko svg olepša za prijaznejši prikaz v brskalniku - omogoči, da se posamezna vozlišča in njihove povezave obarvajo rdeče ob kliku ali prehodu z miško; -s ali –source nastavi izvorno mapo; -e ali –extension nastavi končnico vhodnih datotek; -o ali –output nastavi ime izhodnih datotek dot in svg; -t ali –splitter nastavi ločilo med skupino in elementom, ki ga uporablja –pretty.

```
45 curdir = os.getcwd()
46 ls = os.listdir(srcdir)
47 entries = [];
48
49 if verbose:
50     print 'verbose set to:', verbose
51     print 'cluster set to:', cluster
52     print 'recursive set to:', recursive
53     print 'source directory set to:', srcdir
54     print 'input extension set to:', ext
55     print 'output file set to:', output
56
57 for entry in ls:
58     entryName = os.path.splitext(entry)[0]
59     if os.path.splitext(entry)[1].lower() == ext:
60         entries.append(entry);
61         if verbose:
62             print 'lexing:', entry
63         lexout = open(os.path.join(srcdir, entryName + '
               .lexout'), 'w')
64         subprocess.call(['./lexPLSQL', entry], stdout=
               lexout)
65         lexout.close()
```

Nato izvedemo leksikalno analizo vseh datotek v nastavljeni izvorni mapi z nastavljeno končnico, rezultat katere je datoteka .lexout s funkcijskimi klici za posamezno datoteko.

```
66 graph = open(output+'.dot', 'w')
67 graph.write('strict digraph {\n')
68 graph.write('\tgraph [compound=true];\n')
69 for entry in entries:
70     if verbose:
71         print 'graphing ', srcdir+os.sep+entry
72     string = ""
73     entryName = os.path.splitext(entry)[0]
74     inputfile = open(os.path.join(srcdir, entryName + '.
           lexout'), 'r')
75     if cluster:
76         string += "\tsubgraph \"cluster" + entryName.
               upper() + "\" {\n"
77         string += "\t\tlabel = \"" + entryName.upper()
               + "\"\n"
78         for line in inputfile:
79 #           function calls within function calls
```

```
80              splitted = line.split('(') if recursive else
                    [line.split('(')[0]]
81              for name in splitted:
82                  for entry in entries:
83                      comp = os.path.splitext(entry)[0].
                           lower()
84                      if name.lower().endswith(comp):
85                          string += "\t\t\""+ entryName.
                               lower()+splitter+comp + "\"
                               -> " + '"cluster' + comp.
                               upper() + '"\n'
86                          string += "\t\t\""+ entryName.
                               lower()+splitter+comp + "\" [
                               label=\"" + comp.lower() + "
                               \"]\n"
87          string += "\t}\n"
88          graph.write(string)
89
90      else:
91          for line in inputfile:
92  #           function calls within function calls
93              splitted = line.split('(') if recursive else
                    [line.split('(')[0]]
94              for name in splitted:
95                  for entry in entries:
96                      comp = os.path.splitext(entry)[0].
                           lower()
97                      if name.lower().endswith(comp):
98                          string += "\t\"" + entryName.
                               lower() + "\" -> \"" + comp
                               + "\"\n"
99          graph.write(string)
100     inputfile.close()
101
102 graph.write("}")
103 graph.close()
```

Ko je leksikalna analiza zaključena, začnemo gradnjo usmerjenega grafa samega. Vsaka datoteka s končnico .lexout, ki jo je generiral leksikalni analizator, po vrsticah predstavlja vse funkcijske klice tiste funkcije, procedure ali dinamične strani, ki ji datoteka pripada. Tako program za vsako vrstico doda v graf povezavo od lastnice datoteke do funkcije, ki jo ta vrstica kliče - vendar le, če je klicana funkcija med datotekami z izvorno kodo. Če je bil podan parameter *–clustered*, potem bo

za vsako vhodno datoteko program ustvaril v grafu podgraf, v katerem bodo predstavljeni vsi funkcijski klici lastnice datoteke, vsak funkcijski klic pa ima povezavo na svoj podgraf.

```
104 if verbose:
105     print 'rendering to: '+output+'.svg'
106 subprocess.call(['fdp', output+'.dot', '-Tsvg', '-o'+
    output+'.svg'])
```

S tem je graf končan in zapisan v datoteki dot. Zdaj ga je treba izrisati za prika. Za ta namen uporabimo modul Graphviz *fdp*, ki razporedi vozlišča po "modelu vzmeti". Rezultat je datoteka svg, ki jo lahko odpremo kar v spletnem brskalniku.

```
107 if pretty:
108     if verbose:
109         print 'prettifying to: pretty_'+output+'.svg'
110     import prettysvg
111     prettysvg.prettify(output+'.svg', splitter)
```

Če je bil podan parameter *–pretty*, bo program na koncu ustvarjeni datoteki svg dodal še funkcije Javascript in stil CSS, ki skupaj omogočajo, da se v brskalniku posamezna vozlišča in pripadajoče povezave ob dotiku z miškinim kazalcem ali klikom obarvajo rdeče, kar olajša pregledovanje grafa v brskalniku.

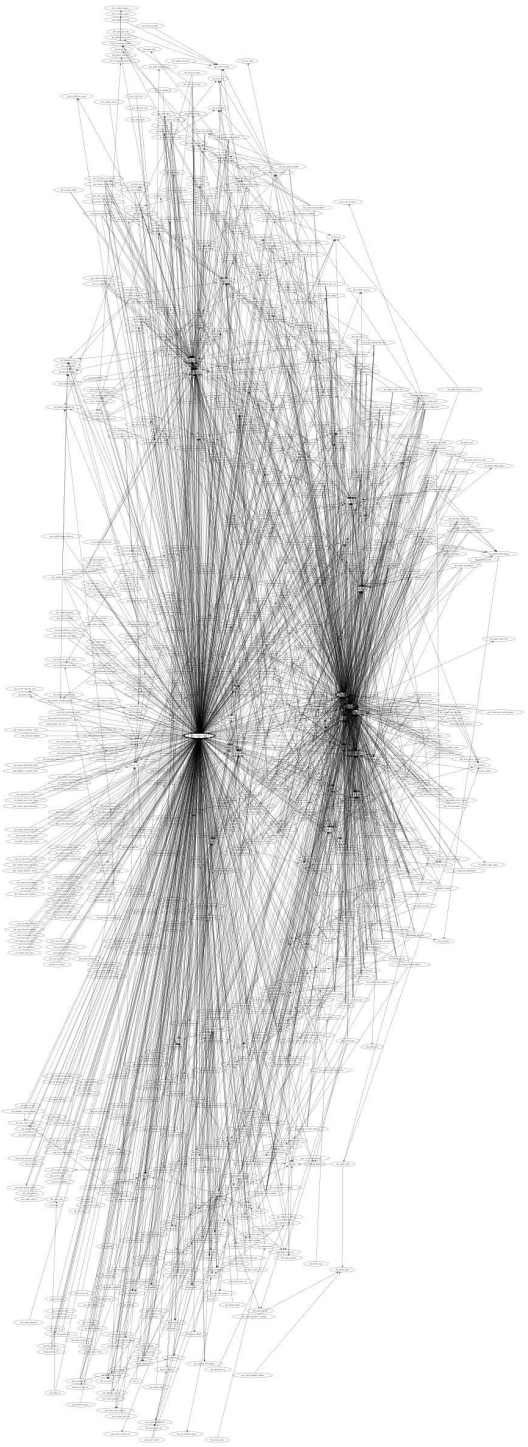
Poglavje 5

Rezultati

Potem ko smo uspešno pridobili datoteke izvorne kode za vsako funkcijo, proceduro in dinamično strukturo v podmapo *SOURCE*, poženemo našo skripto Python (ki mora seveda imeti dostop do vseh povezanih programov in skript) s parametri *–verbose*, *–recursive*, *–pretty* ter *–source=SOURCE* in *–output=recursive_fdp_all*:

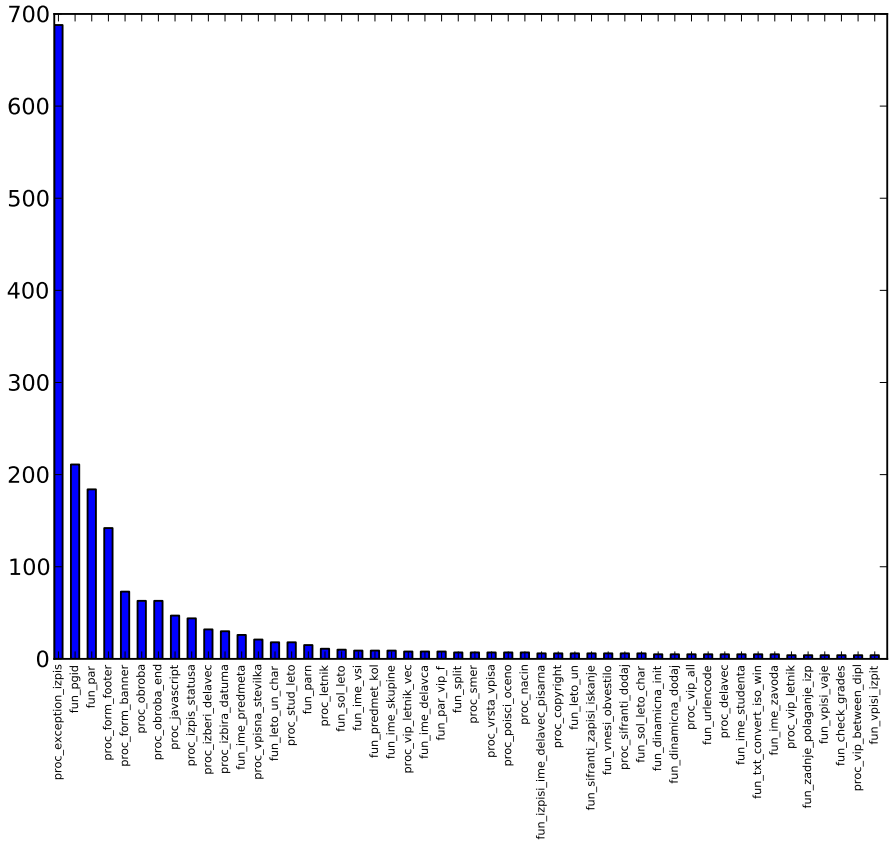
```
python genATG.py -vrp -s SOURCE -o recursive_fdp_all
```

S tem se ustvari datoteka *recursive_fdp_all.dot*, ki predstavlja opis grafa prehodov aplikacije, hkrati se ta opis vizualno upodobi v obliki datoteke *recursive_fdp_all.svg*. Ti datoteki se, ker smo uporabili parameter *–pretty* nato še dodata koda JS in stil CSS, ki skupaj omogočata barvanje posameznih povezav in vozlišč, kar se shrani v datoteko s predpono *pretty_*. Tako pridobljeni graf aplikacije e-Študent, ustvarjen s tem procesom, je prikazan na sliki 5.1. Vsako vozlišče na grafu predstavlja eno funkcijo, proceduro ali dinamično stran, medtem ko vsaka usmerjena povezava iz enega vozlišča na drugo predstavlja en klic izvora na ponor. Če bi vsako vozlišče v grafu predstavljalo CIM z vsemi njegovimi elementi, se pravi če bi vsakemu vozlišču dodali začetno stran, ki ga naslavlja, atomične odseke, ki ga naslavljajo, ter izraz, ki opisuje strukturo strani (vhodne in izhodne povezave namreč že imamo), potem bi dobili ATG, ki bi predstavljal ASM.



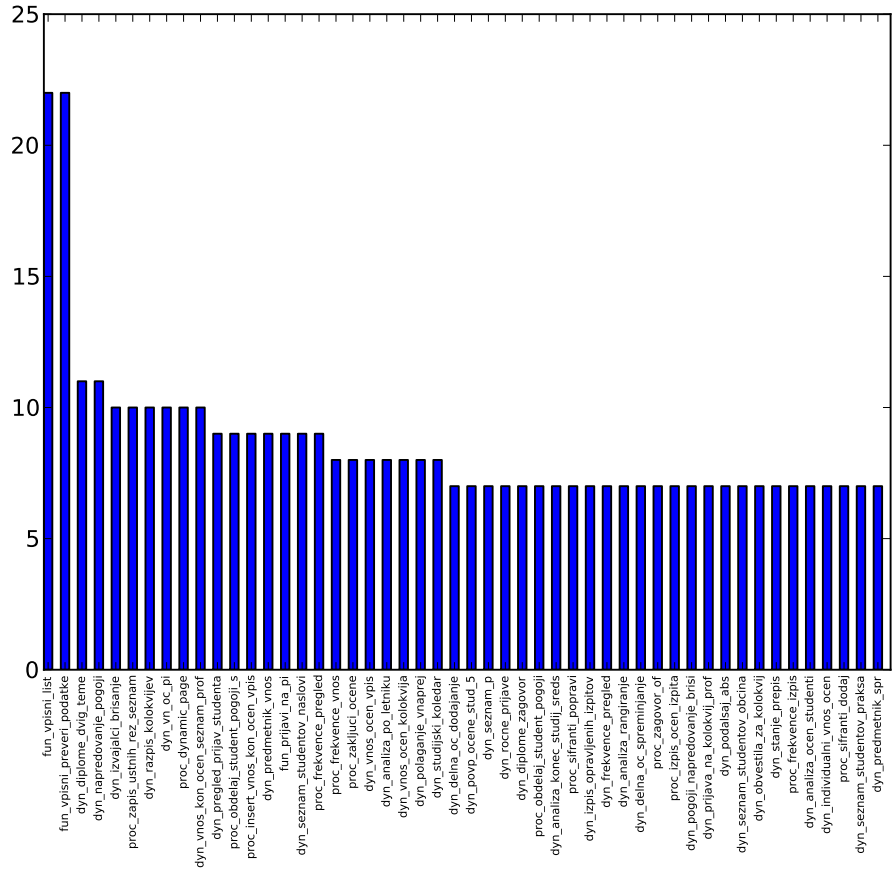
Slika 5.1: Graf dobljen iz vseh funkcij, procedur in dinamičnih struktur

Kot je razvidno, je graf kompleksen in težko pregleden, saj vsebuje 885 vozlišč in 2223 povezav, med katerimi očitno nekateri prevladujejo. Ob gradnji grafa sta se privzeto izračunali tudi vhodna in izhodna stopnja posameznega vozlišča, katerih povprečje znaša 2.51. Ti sta predstavljeni vsaka na svojem grafu - prva z zapono _indeg na sliki 5.2 druga pa z zapono _outdeg na sliki 5.3.



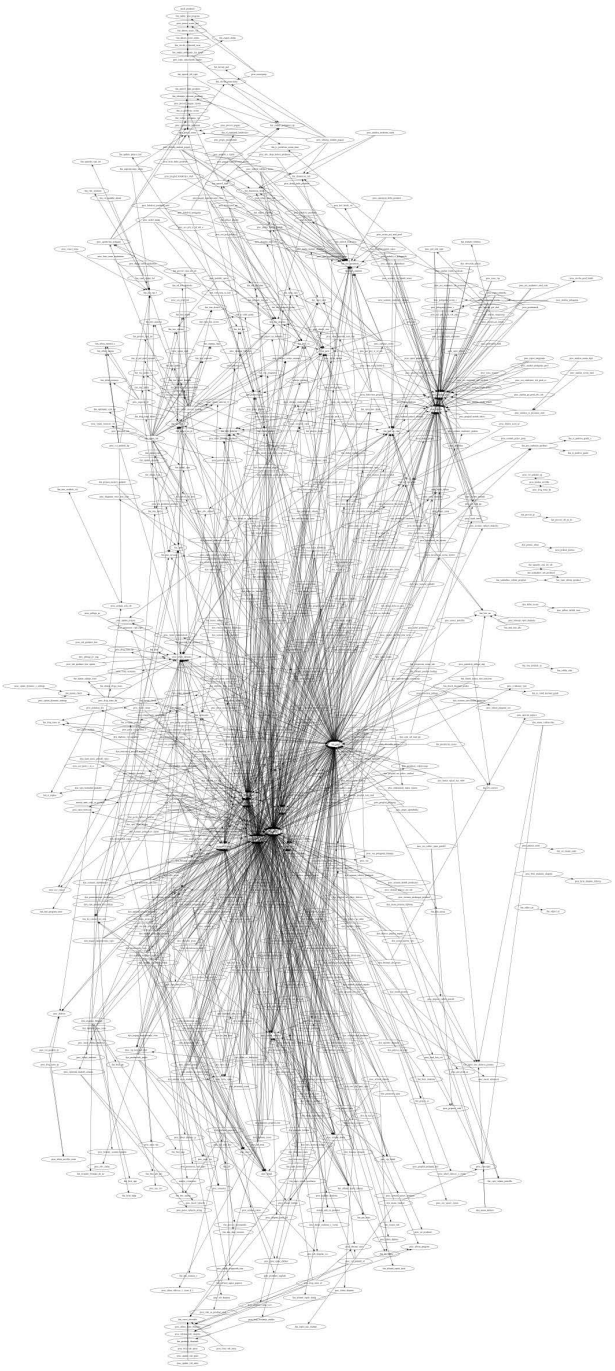
Slika 5.2: Graf petdesetih vozlišč z najvišjo vhodno stopnjo

Na grafih našega primera se opazi, da vhodna stopnja izrazito izstopa pri prvih štirih vozlišč, pri katerih je nad 100. Med temi prednjači vozlišče *proc_exception_izpis*, temu sledijo *fun_pgid*, *fun_par* ter *proc_form_footer*, medtem ko izhodna stopnja ni pretirano velika - največ povezav na druga vozlišča imata vozlišči *fun_vpisni_list* in *fun_vpisni_preveri_podatke*. Zdaj se mora uporabnik sam odločiti kaj bo s temi podatki storil - ali se bodo vozlišča z visokimi stopnjami kritično ovrednotila in



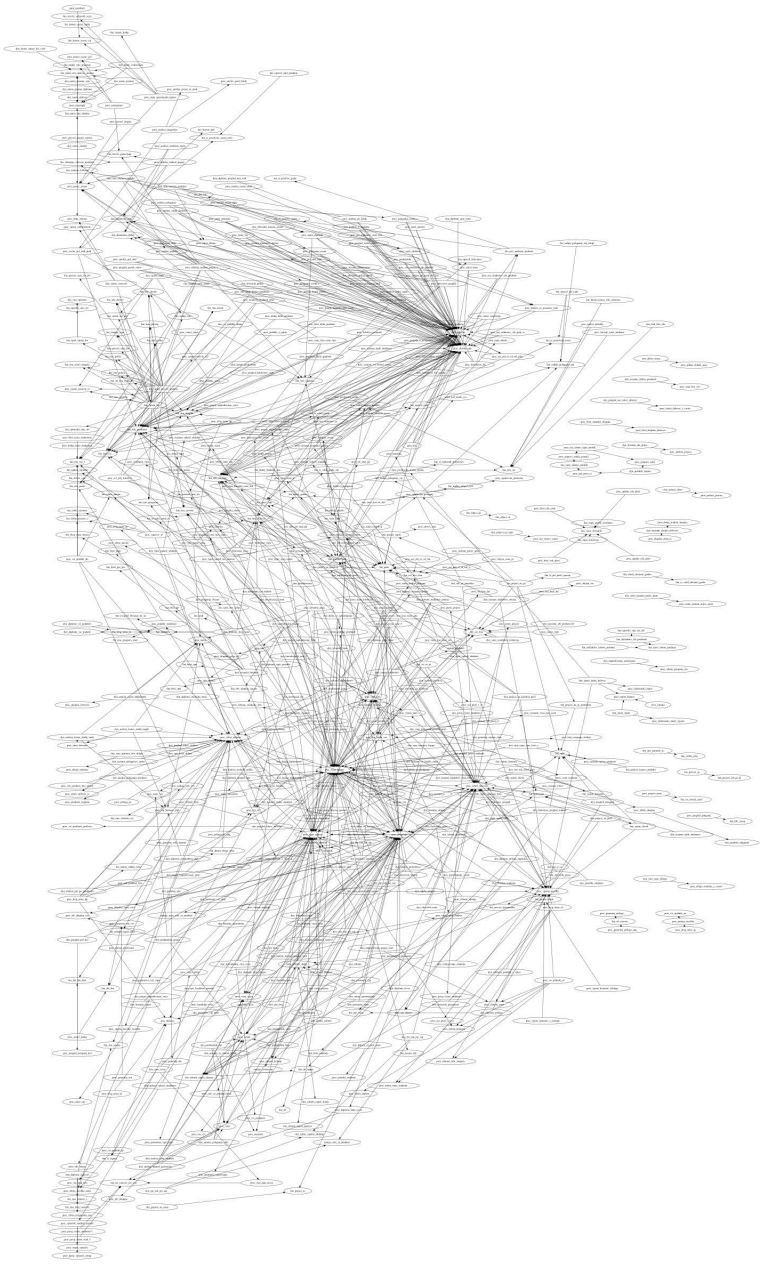
Slika 5.3: Graf petdesetih vozlišč z najvišjo izhodno stopnjo

vzela pod drobnogled, ali pa se bodo enostavno izločila zgolj iz želje odkrivanja drugih zakonitosti, zavedati pa se seveda mora, da se s tem zabriše celotna slika aplikacije in graf lahko postane zavajajoč. Kot primer smo tu najprej predpostavili, da je uporabnik pregledal funkcijo *proc_exception_izpis* in prišel do zaključka, da ta funkcija nima večje povezave s samo logiko aplikacije. Zaradi tega jo je izločil in še enkrat ustvaril graf, ki je predstavljen na sliki 5.4 in ima 653 vozlišč in 1534 povezav, kar pomeni da je povprečna stopnja vozlišč zdaj 2.35.



Slika 5.4: Graf dobljen iz vseh funkcij, procedur in dinamičnih struktur. Izključeno je bilo vozlišče *proc_exception_izpis*.

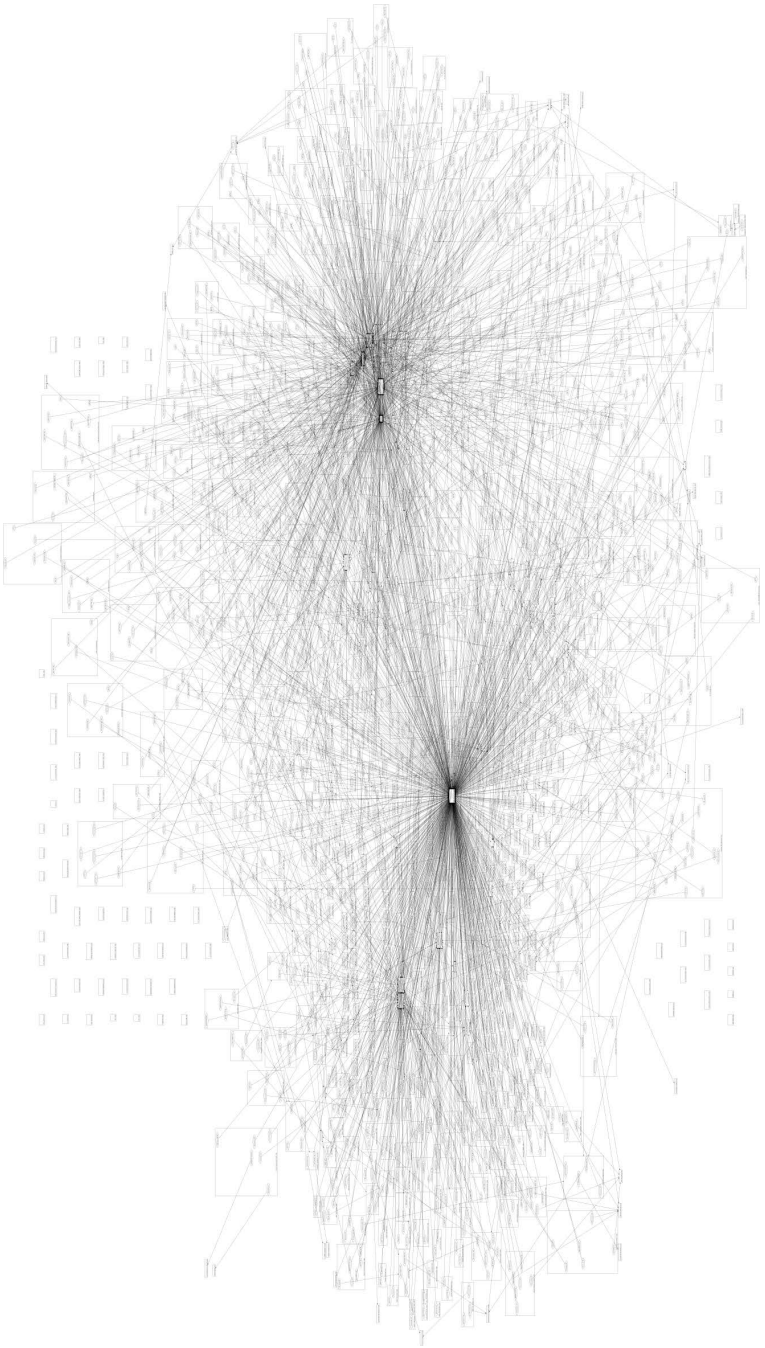
Druga predpostavka bi bila, da se je uporabnik arbitrarno odločil odstraniti vsa vozlišča s stopnjo, večjo od 100, kar pomeni, da so bila v našem primeru odstranjena vozlišča *proc_exception_izpis*, *fun_pgid*, *fun_par* in *proc_form_footer*. Nato je ponovno ustvaril graf, ki je predstavljen na sliki 5.5, in ima 617 vozlišč in 998 povezav. Tak graf je zavoljo povprečne stopnje vozlišč 1.62, manj izredno povezanih vozlišč ter dodane funkcije označevanja mnogo lažje pregledovati in raziskovati, čeprav, kot rečeno, ne prikazuje več stoodstotno verodostojne slike.



Slika 5.5: Graf dobljen iz vseh funkcij, procedur in dinamičnih struktur. Izključena so vozlišča s stopnjo večjo od 100.

Če je v našem programu vklopljen parameter gručenja *–cluster*, so rezultati načeloma enaki, razlikuje se le njihova predstavitev. Zdaj je vsako vozlišče predstavljeno kot skupina, funkcijskih klicev, ki jih to vozlišče vsebuje, posamezna notranja vozlišča pa imajo nato povezave na skupine, ki jih predstavljajo. Razlika je še ta, da so v tem primeru na grafu prikazane tudi skupine brez notranjih vozlišč, torej funkcije, procedure ali dinamične strukture brez klicev, s čimer naraste število vozlišč na 959, število povezav pa ostane 2223.

Slika 5.6: Detajlni izsek grafa 5.7



Slika 5.7: Graf dobljen iz vseh funkcij, procedur in dinamičnih struktur z vključenim parametrom gručenja.

Poglavje 6

Sklepne ugotovitve

Ugotovili smo, da so spletne aplikacije in njihove funkcije, procedure in dinamične strukture zelo kompleksne in na prvi pogled neobvladljive, vendar se z našim orodjem in njemu podobnimi, da omogočiti pogled iz ptičje perspektive in je tega, s posameznimi prijemi, kot je na primer dodajanje poudarkov, mogoče tudi nekoliko olajšati. Na primeru podatkovne baze e-Študent smo pokazali, da je z našim programom mogoče generirati za poljubno shemo PL/SQL graf klicev posameznih funkcij, procedur in dinamičnih struktur, ki je približek ATG. Temu bi sledilo razvrščanje vozlišč grafa v gruče, vendar to ni več tema te diplomske naloge. Načeloma je program generičen do te mere, da bo, če imamo primerne vhodne podatke v njemu prijazni obliki (vsaka funkcija v svoji tekstovni datoteki), brez težav deloval za poljuben programski jezik, ki ima podobno semantiko funkcijskih klicev kot PL/SQL, oziroma je tako modularen, da lahko posamezne module uporabljamo samostojno ali jih tudi zamenjamo.

Literatura

[1] Flex - fast lexical analyzer, dostopno na:
http://flex.sourceforge.net/

[2] Python
https://www.python.org/

[3] Graphviz
http://www.graphviz.org/

[4] Rožanc, I., Slivnik, B.: Using Reverse Engineering to Construct the Platform Independent Model of a Web Application for Student Information Systems. Computer Science and Information Systems, Vol. 10, No. 4, 1557-1583. (2013)